Podstawy Programowania 2

Jednokierunkowa lista liniowa i rekurencja

Arkadiusz Chrobot

Katedra Systemów Informatycznych

31 marca 2020

Plan

Wstęp

Na poprzednich wykładach dowiedzieliśmy się, że budowa typu bazowego takich dynamicznych struktur danych jak stos lub kolejka jest rekurencyjna. Typ ten oparty jest na strukturze, która zawiera pole wskaźnikowe, które może wskazywać inne zmienne tego samego typu co ona. Tymi zmiennymi są kolejne elementy struktury. Ten opis dotyczy także jednokierunkowej listy liniowej, która jest bardziej ogólną strukturą, niż dwie wymienione wcześniej, a zatem wymaga większej liczby operacji podstawowych do zaimplementowania. Powstaje zatem pytanie, czy zrealizowanie ich w postaci funkcji rekurencyjnych nie byłoby bardziej korzystne? Okazuje się, że podejście rekurencyjne upraszcza wiele elementów obsługi listy. Przekonamy się o tym modyfikując program z poprzedniego wykładu tak, aby korzystał on głównie z funkcji rekurencyjnych. Dowiemy się także o związkach rekurencji z programowaniem funkcyjnym i jak możemy taki sposób programowania wykorzystać w języku C.

Założenia

Przypomnijmy, że opisywany program wykorzystuje jednokierunkową listę liniową do przechowywania liczb naturalnych w porządku niemalejącym. Jest to zatem lista uporządkowana. Jej konstrukcja mimo przyjętego założenia nie ogranicza wartości elementów jedynie do liczb naturalnych - mogą one przechowywać każdą liczbę należącą do typu `int`.

Typ bazowy i wskaźnik listy

```
1  #include<stdio.h>
2  #include<stdlib.h>
3
4  struct list_node {
5      int data;
6      struct list_node *next;
7  } *list_pointer;
```

Typ bazowy i wskaźnik listy

Początek programu nie ulega zmianom. Włączane są te same pliki nagłówkowe co poprzednio. Definicja typu bazowego i deklaracja wskaźnika listy również pozostają te same. Nie zmienia się również założenie co do wskaźnika listy - zawsze powinien on wskazywać jej pierwszy element lub mieć wartość `NULL`.

Operacja wstawiania elementu do listy

Jeśli zaczniemy myśleć w sposób rekurencyjny o operacji wstawiania nowego elementu do uporządkowanej jednokierunkowej listy liniowej, to dojdziemy do wniosku, że w tej operacji musimy wyróżnić dwa przypadki:

1. nowy element jest wstawiany od pustej (nieistniejącej) listy,
2. nowy element jest wstawiany do istniejącej (niepustej) listy.

Drugi przypadek obejmuje wszystkie sytuacje związane ze wstawianiem nowego elementu do istniejącej listy uporządkowanej, tj. wstawienie na jej początku, w jej wnętrzu i na końcu. Proszę zwrócić uwagę, że nie w podejściu rekurencyjnym nie jest potrzebna operacja tworzenia listy, bo jest ona ujęta w pierwszym przypadku.
Operacja wstawiania nowego elementu do listy zostanie zrealizowania przy użyciu jednej funkcji „głównej” i jednej pomocniczej. Z nią zapoznamy się najpierw.

Operacja wstawiania elementu do listy

Funkcja `create_and_add_node()`

```
1  int create_and_add_node(struct list_node **list_pointer,
2                          int number)
3  {
4      struct list_node *new_node = (struct list_node *)
5                          malloc(sizeof(struct list_node));
6      if(!new_node)
7          return -1;
8      new_node->data = number;
9      new_node->next = *list_pointer;
10     *list_pointer = new_node;
11     return 0;
12 }
```

Operacja wstawiania elementu do listy

Funkcja `create_and_add_node()`

Funkcja zgodnie ze swoją nazwą tworzy nowy element i wstawia go do listy. Zwraca ona wartość typu `int`, która sygnalizuje status wykonania przez nią opisywanej czynności. Posiada ona również dwa parametry. Pierwszym jest podwójny wskaźnik typu bazowego listy, a drugim jest zmienna typu `int`, przez którą przekazywana do funkcji będzie liczba, jaka ma być umieszczona w nowym elemencie. Proszę zwrócić uwagę, że treść tej funkcji jest stosunkowo prosta. Najpierw jest przydzielana pamięć na nowy element listy (wiersze 4 i 5). Jeśli ten przydział się nie udał funkcja kończy swoje działanie zwracając wartość −1 i tym samym sygnalizując niepowodzenie operacji wstawiania nowego elementu do listy. Jeśli jednak przydział pamięci zakończył się powodzeniem, to do elementu zapisywana jest przekazana liczba (wiersz 8), w jego polu `next` zapisywany jest adres zapisany w zmiennej wskazywanej przez podwójny wskaźnik (wiersz 9), a następnie w tej zmiennej jest umieszczany adres nowego elementu.

Operacja wstawiania elementu do listy

Funkcja `create_and_add_node()`

Funkcja `create_and_add_node()` wykonuje czynności, które odpowiadają w programie z poprzedniego wykładu wszystkim przypadkom wstawienia nowego elementu do uporządkowanej listy oraz dodatkowo operacji utworzenia listy (wstawienia do niej jej pierwszego elementu). Odpowiedź na pytanie dlaczego taka prosta funkcja może obsłużyć wspomniane czynności, które były poprzednio realizowane przy pomocy kilku osobnych funkcji poznamy, gdy zobaczymy sposób jej użycia w funkcji `add_node()` odpowiedzialnej za ustalenie miejsca na liście gdzie ma być wstawiony nowy element i wywołanie opisywanej funkcji.

Operacja wstawiania elementu do listy

Funkcja add_node()

```
1  int add_node(struct list_node **list_pointer, int number)
2  {
3      if(*list_pointer!=NULL && (*list_pointer)->data<number)
4          return add_node(&(*list_pointer)->next,number);
5      else
6          return create_and_add_node(list_pointer,number);
7  }
```

Operacja wstawiania elementu do listy

Funkcja `add_node()`

Funkcja `add_node()` z dokładnością do nazwy ma taki sam prototyp jak `create_and_add_node()`. Znaczenie zwracanej przez nią wartości oraz drugiego parametru również jest takie samo. Zmienia się natomiast rola podwójnego wskaźnika. Przy pierwszym wywołaniu funkcji do tego wskaźnika zapisywany jest adres wskaźnika listy, zatem wszelkie zmiany wykonane na tym parametrze będą również wykonane na wskaźniku listy. Jeśli dojdzie do wywołania rekurencyjnego funkcji, to w tym parametrze znajdzie się adres pola `next` elementu wskazywanego przez wskaźnik, którego adres był zapisany w parametrze w poprzednim wywołaniu funkcji (wiersz nr 4). Prześledźmy możliwe scenariusze wykonania tej funkcji:

Wstawienie elementu do pustej listy.

Wyrażenie `*list_pointer!=NULL` z warunku zapisanego w wierszu nr 3 będzie fałszywe, zatem od razu w wierszu nr 6 zostanie wywołana funkcja `create_and_add_node()`, która utworzy i doda nowy i zarazem pierwszy i jedyny element listy.

Operacja wstawiania elementu do listy

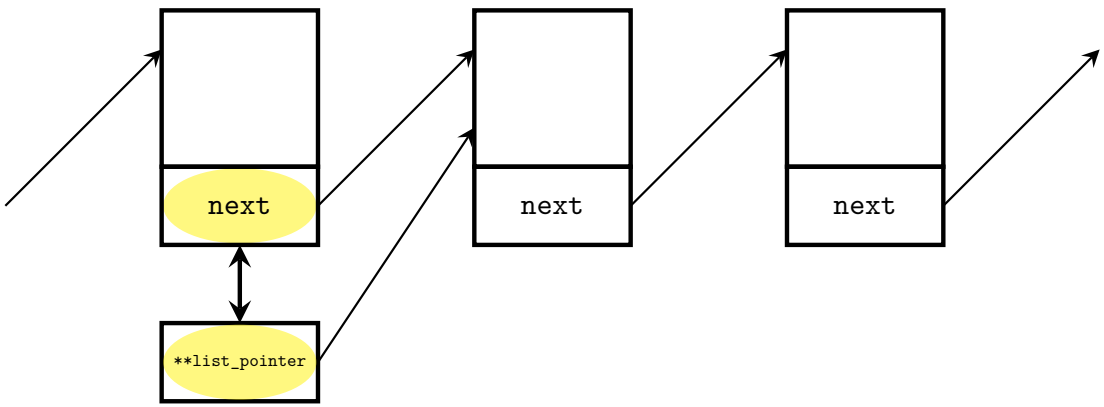
Funkcja `add_node()`

Wstawienie elementu na początek listy.
W tym przypadku pierwsze wyrażenie w warunku z wiersza nr 3 będzie prawdziwe, ale fałszywe będzie drugie (to po operatorze &&), zatem znowu od razu zostanie wywołana `create_and_add_node()`, w wierszu nr 6, która tym razem doda element na początku listy.

Wstawienie elementu wewnątrz listy.
Tym razem oba wyrażenia w warunku z wiersza nr 3 będą prawdziwe i funkcja wywoła się rekurencyjnie (wiersz nr 4). Te wywołania będą tak długo występowały, dopóki drugie wyrażenie we wspomnianym warunku nie stanie się fałszywe. Będzie to znaczyło, instancja (wywołanie) funkcji, dla którego taka sytuacja wystąpiła powinna wstawić nowy element, przed elementem wskazywanym przez zmienną, które adres jest z kolei zapisany w parametrze `list_pointer`. Tą zmienną jest pole `next` elementu poprzedzającego ten element, przed którym ma być wstawiony nowy. Relację między nimi wyjaśnia ilustracja na następnym slajdzie.

Operacja wstawiania elementu do listy

Związek między `struct list_node **list_pointer`, a polem `next`



Ilustracja związku między podwójnym wskaźnikiem, a wskazywanym przez niego polem `next`

Operacja wstawiania elementu do listy

Związek między `struct list_node **list_pointer`, a polem `next` - komentarz

Zaznaczenie pola `next` i wskaźnika `**list_pointer` na rysunku żółtą elipsą oznacza, że te dwie zmienne należy rozpatrywać jako jedną, tzn. modyfikacja wartości jednej z nich pociąga za sobą natychmiastową modyfikację drugiej.

Operacja wstawiania elementu do listy

Funkcja `add_node()`

Wstawienie elementu wewnątrz listy - c.d.
Utworzeniem i wstawieniem elementu dla tego przypadku znowu zajmuje się funkcja `create_and_add_node()` wywoływana w wierszu nr 6.
Wstawienie elementu na końcu listy.
W tym przypadku po serii wywołań rekurencyjnych wystąpi instancja (wywołane) funkcji, dla której pierwsze wyrażenie z warunku w wierszu nr 3 nie będzie spełnione. Będzie to oznaczać, że należy nowy element wstawić na końcu istniejącej listy. Tak, jak w pozostałych przypadkach zajmie się tym funkcja `create_and_add_node()` wywołana w wierszu nr 6 funkcji `add_node()`.

Operacja usuwania elementu z listy

Operacja usuwania elementu z uporządkowanej jednokierunkowej listy liniowej w wydaniu rekurencyjnym polega po prostu na znalezieniu w liście elementu o zadanej wartości i jego usunięciu. Jej przebieg jest zawsze taki sam, niezależnie od tego, gdzie w liście znajduje się element. Przypomnijmy dla porządku, że jeśli w liście znajduje się więcej niż jeden element o zadanej wartości, to usunięcie pierwszego z nich spełnia założenia tej operacji. Na następnym slajdzie znajduje się kod źródłowy funkcji rekurencyjnej realizującej usunięcie elementu z listy. Jej kod jest na tyle krótki w zapisie, że nie wymaga on rozbicia na funkcje pomocnicze.

Operacja usuwania elementu z listy

Funkcja delete_node()

```
1  void delete_node(struct list_node **list_pointer, int number)
2  {
3      if(*list_pointer) {
4          if((*list_pointer)->data == number) {
5              struct list_node *next = (*list_pointer)->next;
6              free(*list_pointer);
7              *list_pointer = next;
8          } else
9              delete_node(&(*list_pointer)->next, number);
10     }
11 }
```

Operacja usuwania elementu z listy

Funkcja `delete_node()`

Opisywana funkcja nic nie zwraca, ponieważ efekty jej działania nie generują sytuacji wyjątkowych i są widoczne po wyświetleniu listy na ekranie. Funkcja ma dwa parametry - pierwszy jest podwójnym wskaźnikiem typu `struct list_node`, a drugi typu `int`. Przez ten ostatni do funkcji przekazywana jest liczba, jaką powinien mieć zapisaną element do usunięcia. Przy pierwszym wywołaniu funkcji sprawdzane jest najpierw, czy przekazana jej lista nie jest pusta (wiersz nr 3). Jeśli ona istnieje, to w wierszu nr 4 następuje sprawdzenie, czy pierwszy element tej listy nie zawiera liczby przekazanej przez drugi parametr funkcji. Jeśli tak jest to powinien on zostać usunięty, dlatego najpierw zapamiętywany jest adres przechowywany w polu `next` tego elementu, następnie zwalniana jest pamięć na ten element i we wskaźniku `*list_pointer` zapamiętywany jest adres elementu poprzednio wskazywanego przez pole `next` usuniętego elementu listy.

Operacja usuwania elementu z listy

Funkcja `delete_node()`

Jeśli jednak warunek z wiersza nr 4 nie jest spełniony dla pierwszego elementu, to funkcja wywołuje się rekurencyjnie (wiersz nr 9), celem znalezienia takiego elementu listy, który by go spełniałby. Proszę zwrócić uwagę, że za pierwszy parametr tej funkcji w wywołaniu rekurencyjnym jest podstawiany adres pola `next` elementu badanego w danej instancji funkcji. Zatem jeśli będą wykonane w kolejnym wywołaniu rekurencyjnym funkcji modyfikacje wartości tego parametru, to zostaną one natychmiast odzwierciedlone w tym polu. W ten sposób instrukcje z wierszy nr 5, 6 i 7 obsługują również przypadek usunięcia elementu z wnętrza listy i z końca listy. Jeśli funkcja w wyniku kolejnych wywołań rekurencyjnych nie znajdzie elementu o wartości wskazującej, że ma być on usunięty, to zostanie wywołana rekurencyjnie dla pola `next` o wartości `NULL`. W takim wypadku nic nie wykona, tylko się zakończy, podobnie jak jej wcześniejsze wywołania rekurencyjne. W ten sposób obsługiwany jest przypadek, kiedy nie ma elementu na liście, który spełniałby kryteria usunięcia.

Operacja wypisania zawartości listy

Operacja wypisania zawartości listy w wydaniu rekurencyjnym jest równie prosta, jak w wydaniu iteracyjnym. Można ją opisać krótko:

1. jeśli lista istnieje wypisz zawartość jej pierwszego elementu,
2. wypisz zawartość reszty listy.

Kolejny slajd zawiera kod źródłowy funkcji, która realizuje tę operację.

Operacja wypisania zawartości listy

Funkcja `print_list()`

```
1  void print_list(struct list_node *list_pointer)
2  {
3      if(list_pointer) {
4          printf("%d ",list_pointer->data);
5          print_list(list_pointer->next);
6      } else
7          puts("");
8  }
```

Operacja wypisania zawartości listy

Funkcja `print_list()`

Funkcja `print_list()` nie zwraca żadnej wartości - w jej przypadku nie jest to konieczne. Przyjmuje ona jeden argument wywołania, jakim jest wskaźnik na listę. W wierszu nr 3 funkcja sprawdza, czy przekazany jej wskaźnik nie jest pusty. Jeśli ten warunek jest spełniony, to wypisuje wartość pola `data` elementu listy wskazywanego przez przekazany jej wskaźnik, a następnie wywołuje się rekurencyjnie podstawiając za swój parametr adres zawarty w polu `next` bieżącego elementu. Jeśli ten adres nie ma wartości `NULL`, to znaczy że istnieje kolejny element listy i kolejna instancja funkcji powtórzy czynności z wierszy nr 4 i 5. Wywołanie rekurencyjne, które zostanie uruchomione dla wartości pola `next` równej `NULL` przeniesie kursor do następnego wiersza ekranu, korzystając z funkcji `puts()` i zakończy się, a po nim wcześniejsze wywołania rekurencyjne funkcji `print_list()`.

Operacja wypisania zawartości listy w odwrotnej kolejności

Funkcja `print_list_inversely()`

Okazuje się, że niewielka modyfikacja funkcji `print_list()` umożliwia wykonanie operacji, która w wersji iteracyjnej tej funkcji była bardzo trudna do uzyskania - wypisanie wartości elementów listy w odwrotnej kolejności. Wystarczy zamienić kolejność wierszy nr 3 i 4, tak aby wartość elementu była wypisywana na ekranie *po powrocie* z wywołania rekurencyjnego. Należy także zrezygnować z użycia funkcji `puts()` wewnątrz funkcji wypisującej elementy listy, ponieważ musiałaby ona być wywołana po wypisaniu wartości pierwszego elementu listy, a to jest trudne do wykrycia. Można kursor przenieść do kolejnego wiersza ekranu po zakończeniu tej funkcji. Na następnym slajdzie zamieszczony jest kod funkcji `print_list_inversely()`, który zawiera opisane zmiany.

Operacja wypisania zawartości listy w odwrotnej kolejności

Funkcja `print_list_inversely()`

```
1  void print_list_inversely(struct list_node *list_pointer)
2  {
3      if(list_pointer) {
4          print_list_inversely(list_pointer->next);
5          printf("%d ",list_pointer->data);
6      }
7  }
```

Usunięcie list

Operacja usunięcia wszystkich elementów listy przebiega w wydaniu rekurencyjnym bardzo podobnie do wypisania listy zawartości listy w odwrotnym kierunku. W implementacji różnica polega na innej deklaracji parametru funkcji (tym razem jest to podwójny wskaźniki) i wykonywanej na elemencie operacji. Następny slajd zawiera kod źródłowy funkcji `remove_list()`, która realizuje tę operację.

Usunięcie list

Funkcja `remove_list()`

```
1  void remove_list(struct list_node **list_pointer)
2  {
3      if(*list_pointer) {
4          remove_list(&(*list_pointer)->next);
5          free(*list_pointer);
6          *list_pointer = NULL;
7      }
8  }
```

Usunięcie list

Funkcja `remove_list()`

Funkcja ta nie zwraca żadnej wartości, ale posiada jeden parametr, który jest podwójnym wskaźnikiem typu `struct list_node`. Podczas pierwszego wywołania funkcja sprawdza w trzecim wierszu, czy przekazana jej lista istnieje. Jeśli tak, to wywołuje się ona rekurencyjnie w wierszu nr 4. Jej kolejne wywołania rekurencyjne następują tak długo aż któreś z nich zostanie wywołane dla pola `next` ostatniego elementu listy. To pole ma wartość `NULL`, zatem instancja funkcji dla tego pola nic nie wykona i zakończy się. Sterowanie wróci do instancji, która była wywołana dla ostatniego elementu listy, a konkretnie do wiersza nr 5 tej instancji. Tu nastąpi zwolnienie pamięci przydzielonej na ostatni element, nadanie wartości `NULL` polu `next` elementu, który go poprzedza i zakończenie tego wywołania, a tym samym powrót do instancji funkcji wywołanej dla przedostatniego elementu, która powtórzy dla niego opisane wyżej czynności. Te powroty będą następowały tak długo, aż zostanie zwolniona pamięć na pierwszy element listy i zakończy się instancja funkcji wywołana dla niego.

Usunięcie list

Funkcja `remove_list()`

Proszę zwrócić uwagę, że wiersze 4 i 5 opisywanej funkcji nie mogą być zamienione miejscami, bo funkcja byłaby wywoływana rekurencyjnie dla nieistniejącego elementu listy.

Podejście funkcyjne

Rekurencja jest bardzo silnie związana z funkcyjnym paradygmatem programowania, gdzie dosyć często zastępuje iterację. Nie jest ona jednak jedynym jego elementem. W modelu funkcyjnym najważniejszym pojęciem jest funkcja. Zmienne po przypisaniu wartości nie zmieniają jej, zatem podprogramy na nich operujące nie mają efektów ubocznych. Funkcje mogą być przekazywane przez parametry do innych funkcji, które wykonują przy ich pomocy określone operacje lub wykonują te operacje na przekazanych im funkcjach. Te inne funkcje określa się mianem *funkcji wyższego rzędu* (ang. *higher order functions*). Język C nie wspiera bezpośrednio funkcji wyższego rzędu, ani paradygmatu funkcyjnego, ale korzystając ze wskaźników możemy uzyskać przybliżony efekt.

Wykonywanie operacji bez wyników na liście

Na początku napiszemy funkcję, która rekurencyjnie „odwiedza” każdy z elementów listy i wykonuje operację, której definicja jest określona przez inną funkcję, na którą wskaźnik jest jej przekazywany jako argument wywołania. Następny slajd zawiera definicję tej funkcji.

Wykonywanie operacji bez wyników na liście

Funkcja `iterate_list()`

```
1  void iterate_list(struct list_node *list_pointer,
2                            void (*action)(struct list_node *))
3  {
4      if(list_pointer) {
5          if(action)
6              action(list_pointer);
7          iterate_list(list_pointer->next,action);
8      }
9  }
```

Wykonywanie operacji bez wyników na liście

Funkcja `iterate_list()`

Funkcja ta nie zwraca żadnej wartości, ale posiada dwa parametry. Przez pierwszy przekazywany będzie wskaźnik na listę, a przez drugi wskaźnik na funkcję, która również nie zwraca żadnej wartości, ale przyjmuje jako argument wywołania wskaźnik na element listy, na którym ma wykonać określoną operację. Funkcja `iterate_list()` jest funkcją wyższego rzędu. W wierszu nr 4 sprawdza ona, czy wskaźnik na element listy, który jej został przekazany jest niepusty. Jeśli ten warunek jest spełniony, to sprawdza, czy wskaźnik na funkcję również jest niepusty. W przypadku gdyby tak nie było, to nie zostaną wykonane żadne operacje dla bieżącego elementu listy, jedynie funkcja `iterate_list()` wywoła się rekurencyjnie dla kolejnego elementu (wiersz 7). Jeśli jednak wskaźnik `action` będzie miał wartość różną od `NULL`, to wskazywana przez niego funkcja zostanie wywołana dla elementu bieżąco wskazywanego przez pierwszy parametr funkcji `iterate_list()`.

Wykonywanie operacji bez wyników na liście

Funkcja `print_element()`

```
1  void print_element(struct list_node *list_pointer)
2  {
3      printf("%d ",list_pointer->data);
4  }
```

Wykonywanie operacji bez wyników na liście

Funkcja `print_element()`

Funkcja `print_element()` realizuje przykładową operację na pojedynczym elemencie listy - wypisuje na ekran wartość jego pola `data`. Zatem, jeśli funkcja `iterate_list()` zostanie wywołana ze wskaźnikiem na funkcję `print_element()` przekazanym jej jako drugi argument wywołania, to wypisze ona na ekran wartości wszystkich elementów listy. Różnica między jej działaniem, a `print_list()` będzie polegała na tym, że nie przeniesie ona kursora do następnego wiersza na ekranie.

Wykonywanie operacji bez wyników na liście

Funkcja `double_element_value()`

```
1  void double_element_value(struct list_node *list_pointer)
2  {
3      list_pointer->data*=2;
4  }
```

Wykonywanie operacji bez wyników na liście

Funkcja `double_element_value()`

Ta funkcja, wywołana z poziomu `iterate_list()` podwoi wartość pola `data` elementu listy wskazywanego przez jej parametr. W związku z tym `iterate_list()` podwoi wartość wszystkich elementów listy.

Wykonywanie operacji z wynikami na liście

Przedstawiona realizacja funkcji wyższego rzędu pozwala przeprowadzić operacje, które zmieniają wartości elementów listy, czego przykładem jest funkcja `double_element_value()`. Nie jest to zgodne z modelem programowania funkcyjnego, w którym raz zapisane zmienne nie powinny zmieniać swojej wartości. Spróbujemy zatem dodać do programu inną funkcję wyższego rzędu, która będzie zwracała wartość operacji wykonanej na liście przez funkcje wywołane z jej poziomu. Jej kod źródłowy przedstawiony jest na następnym slajdzie.

Wykonywanie operacji z wynikami na liście

Funkcja `iterate_list_with_result()`

```
1  int iterate_list_with_result(struct list_node *list_pointer,
2          int (*action)(int result, struct list_node *list_pointer))
3  {
4      int result=0;
5      for(; list_pointer; list_pointer=list_pointer->next)
6          if(action)
7              result=action(result,list_pointer);
8      return result;
9  }
```

Wykonywanie operacji z wynikami na liście

Funkcja `iterate_list_with_result()`

Opisywana funkcja ma kod źródłowy bardzo podobny do `iterate_list()`. W przeciwieństwie do tej ostatniej przyjmuje ona przez drugi parametr wskaźnik na funkcję, która zwraca wartość typu `int` oraz posiada dwa parametry. Pierwszym jest zmienna, która będzie akumulowała dotychczasowe wyniki operacji wykonywanej przez tę funkcję na elementach listy. Taki parametr nie byłby potrzebny w przypadku języków wspierających model funkcyjny programowania, ale w języku C jest konieczny. Drugi parametr to wskaźnik na element listy, na którym dana operacja ma być wykonana. Treść funkcji również uległa zmianie. Zamiast rekurencji wykorzystuje ona iterację (wiersz nr 5) do „przemieszczania się" po liście. Dzięki temu zmienna lokalna `result` zadeklarowana w wierszu nr 4 funkcji może być użyta w wierszu nr 7 do zapamiętania wyniku poprzednich operacji wykonanych na pojedynczym elemencie listy. Aby było to możliwe, w tym samym wierszu musi ona być także przekazana jako pierwszy argument wywołania funkcji wskazywanej przez `action`.

Wykonywanie operacji z wynikami na liście

Funkcja add_up()

```
1  int add_up(int result, struct list_node *list_pointer)
2  {
3      return result+list_pointer->data;
4  }
```

Wykonywanie operacji z wynikami na liście

Funkcja `add_up()`

Funkcja zamieszczona na poprzednim slajdzie jest przykładem funkcji, która może być wywołana z poziomu `iterate_list_with_result()`. Jeśli tak się stanie, to ta ostania zwróci wartość sumy wszystkich elementów listy. Będzie ona poprawna, jeśli zmieści się w zakresie wartości typu `int`.

Wykonywanie operacji z wynikami na liście

Funkcja `count_elements()`

```
1  int count_elements(int result, struct list_node *list_pointer)
2  {
3      return result+1;
4  }
```

Wykonywanie operacji z wynikami na liście

Funkcja `count_elements()`

Jeśli funkcja, której kod znajduje się na poprzednim slajdzie zostanie wywołana z poziomu `iterate_list_with_result()`, to ta ostatnia zwróci liczbę elementów listy. Będzie to poprawna wartość, jeśli zmieści się w górnej granicy wartości typu `int`.

Wykonywanie operacji z wynikami na liście

Podsumowanie

Proszę zwrócić uwagę, że przedstawione rozwiązanie ogranicza możliwe do wykonania operacje tylko do takich, które zwracają wartość typu `int` lub kompatybilnego. Typowe języki funkcyjne nie mają takich ograniczeń, ponieważ zazwyczaj są dynamicznie typowane, co oznacza, że programista nie musi określać typów zmiennych, parametrów i wartości zwracanych przez funkcję. Są one dynamicznie wyznaczane w trakcie jej wykonania.

Funkcja `main()`

W głównej funkcji programu zostaną użyte wszystkie funkcje, które zostały wcześniej zdefiniowane. Tak jak w programie z poprzedniego wykładu, ich działanie zostanie przetestowane dla wszystkich ważnych przypadków.

Funkcja main()

Część pierwsza

```
1  int main(void)
2  {
3      int i;
4      for(i=1; i<5; i++)
5          if(add_node(&list_pointer,i)==-1)
6              fprintf(stderr,"Błąd dodawania elementu do listy!\n");
7      for(i=6; i<10; i++)
8          if(add_node(&list_pointer,i)==-1)
9              fprintf(stderr,"Błąd dodawania elementu do listy!\n");
10     print_list(list_pointer);
```

Funkcja `main()`

Część pierwsza

W pierwszej części funkcji `main()` zaprezentowanej na poprzednim slajdzie tworzona jest lista, która będzie zawierała liczby naturalne od `1` do `4` i od `6` do `9`. Proszę zwrócić uwagę na to, że cała ta czynność odbywa się jedynie z użyciem funkcji `add_node()`. Przy każdym jej wywołaniu sprawdzana jest wartość przez nią zwracana. Jeśli byłaby on równia `-1` to program wypisałby na ekranie informację o błędzie dodawania elementu do listy. Proszę zwrócić uwagę, na pierwszy argument wywołania funkcji `add_node()`. Jest to adres wskaźnika listy. Po utworzeniu listy jej zawartość jest wypisywana na ekranie przy pomocy funkcji `print_list()`.

Funkcja main()

Część druga

```
1       if(add_node(&list_pointer,0)==-1)
2           fprintf(stderr,"Błąd dodawania elementu do listy!\n");
3       print_list(list_pointer);
4       if(add_node(&list_pointer,5)==-1)
5           fprintf(stderr,"Błąd dodawania elementu do listy!\n");
6       print_list(list_pointer);
7       if(add_node(&list_pointer,7)==-1)
8           fprintf(stderr,"Błąd dodawania elementu do listy!\n");
9       print_list(list_pointer);
10      if(add_node(&list_pointer,10)==-1)
11          fprintf(stderr,"Błąd dodawania elementu do listy!\n");
12      print_list(list_pointer);
```

Funkcja `main()`

Część druga

W drugiej części funkcji `main()` dodawane są pojedyncze elementy do istniejącej listy tak, aby operacja dodawania dotyczyła początku, środka i jej końca. Dodawany jest także element o wartości takiej, jaka już jest umieszczona w liście. W każdym przypadku sprawdzana jest wartość zwrócona przez funkcję `add_node()`. Po każdym dodaniu nowego elementu zawartość listy jest wypisywana na ekranie przy pomocy funkcji `print_list()`.

Funkcja main()

Część trzecia

```
1       print_list_inversely(list_pointer);
2       puts("");
3       delete_node(&list_pointer,0);
4       print_list(list_pointer);
5       delete_node(&list_pointer,1);
6       print_list(list_pointer);
7       delete_node(&list_pointer,1);
8       print_list(list_pointer);
9       delete_node(&list_pointer,4);
10      print_list(list_pointer);
11      delete_node(&list_pointer,7);
12      print_list(list_pointer);
13      delete_node(&list_pointer,10);
14      print_list(list_pointer);
```

Funkcja `main()`

Część trzecia

W trzeciej części funkcji `main()` lista jest wypisywana w odwrotnym porządku przy pomocy wywołania funkcji `print_list_inversely()`. Kursor przenoszony jest do następnego wiersza ekranu po zakończeniu tej funkcji w wierszu nr 2 opisywanego fragmentu. Dalej, z listy są usuwane kolejne elementy, tak aby operacja ta dotyczyła początku, wnętrza i końca listy. Podejmowana jest też próba usunięcia elementu, którego nie ma na liście (wiersz nr 7), oraz elementu o wartości, która występuje na liście dwukrotnie (wiersz nr 11). Za każdym razem zawartość listy jest wypisywana w całości na ekranie.

Funkcja main()

Część czwarta

```
1       iterate_list(list_pointer,NULL);
2       iterate_list(list_pointer,print_element);
3       puts("");
4       iterate_list(list_pointer,double_element_value);
5       iterate_list(list_pointer,print_element);
6       puts("");
7       printf("Suma wartości elementów listy: %d\n",
8                   iterate_list_with_result(list_pointer,add_up));
9       printf("Liczba elementów listy: %d\n",
10            iterate_list_with_result(list_pointer,count_elements));
11      remove_list(&list_pointer);
12      return 0;
13  }
```

Funkcja `main()`

Część czwarta

Czwarty fragment funkcji `main()` zawiera wywołania funkcji nazywanych w modelu programowania funkcyjnego funkcjami wyższego rzędu. Najpierw jest wywoływana funkcja `iterate_list()`, do której nie przekazano wskaźnika na żadną funkcję wykonującą operację na liście. Pozwala to sprawdzić, czy `iterate_list()` zachowuje się poprawnie. Następnie zawartość listy jest wypisywana na ekran za pomocą wywołania `iterate_list()` ze wskaźnikiem na funkcję wypisującą pojedynczy element listy na ekran. Proszę zwrócić uwagę, że kursor jest przenoszony do nowego wiersza na ekranie po zakończeniu tej funkcji. W wierszu czwartym ta sama funkcja jest wywoływana ze wskaźnikiem na funkcję `double_element_value()`. Jej działanie powoduje tym razem podwojenie wartości każdego elementu listy. Następnie zwartość listy ponownie jest wypisywana na ekranie w ten sam sposób, co poprzednio (wiersze nr 5 i 6).

Funkcja `main()`

Część czwarta

Potem testowane jest działanie funkcji `iterate_list_with_result()`. Najpierw jest ona wywoływana ze wskaźnikiem na funkcję `add_up()`, a następnie ze wskaźnikiem na funkcję `count_element()`. W pierwszym przypadku zsumuje ona wartości elementów listy, a w drugim policzy ich liczbę. Na koniec lista jest usuwana.

Podsumowanie

Zastosowanie rekurencji pozwoliło znacznie skrócić i uprościć zapis funkcji implementujących operacje na jednokierunkowej liście liniowej. W przypadku operacji wypisania zawartości takiej listy w odwrotnej kolejności rekurencja umożliwiła wręcz jej przeprowadzenie. Te przykłady pozwalają stwierdzić, że technika rekursji powinna być znana każdemu dobremu programiście i każdej dobrej programistce.

W ramach wykładu podano także przykłady wykorzystania elementów programowania funkcyjnego do wykonywania operacji na liście. Ten model programowania zyskuje ostatnio na znaczeniu ze względu na to, że pozwala uniknąć wielu problemów w zastosowaniach związanych z przetwarzaniem współbieżnym, które występują w paradygmacie imperatywnym (i tym samym w modelach proceduralnych, strukturalnych i obiektowych). Język C nie wspiera wprost tego paradygmatu, więc zaprezentowane rozwiązania nie są „czysto” funkcyjne, ale mimo to warto się z nimi zapoznać.

Pytania

?

KONIEC

Dziękuję Państwu za uwagę!